# Modeling Computation and Communication Performance of Parallel Scientific Applications: A Case Study of the IBM SP2

Eric L. Boyd, Gheith A. Abandah, Hsien–Hsin Lee, and Edward S. Davidson
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122
PHONE: 313-936-2917; FAX: 313-763-4617
{boyd, gabandah, linear, davidson}@eecs.umich.edu

## Abstract

A methodology for performance analysis of Massively Parallel Processors (MPPs) is presented. The IBM SP2 and some key routines of a finite element method application (FEMC) are used as a case study. A hierarchy of lower bounds on run time is developed for the POWER2 processor, using the MACS methodology developed in earlier work for uniprocessors and vector processors. Significantly, this hierarchy is extended to incorporate the effects of the memory hierarchy of each SP2 node and communication across the High Performance Switch (HPS) linking the nodes of the SP2. The performance models developed via this methodology facilitate explaining performance, identifying performance bottlenecks, and guiding application code improvements.

## 1. Introduction

Scientific applications are typically dominated by loop code, floating–point operations, and array references. The performance of such applications on scalar and vector uniprocessors has been found to be well characterized by the MACS model, a hierarchical series of performance bounds. As depicted in Figure 1, the M bound models the peak floating–point performance of the *M*achine architecture independent of the application. The MA bound models the machine in conjunction with the operations deemed to be essential in the high–level *A*pplication workload. Hence Gap A represents the performance degradation due to the application algorithm's need for operations that are not entirely masked by the most efficient class of floating–point operations. The MAC bound is derived from the actual *C*ompiler–generated workload. Hence Gap C represents the performance degradation due to the additional operations introduced by the compiler. The MACS bound factors in the compiler–generated *S*chedule for the workload. Gap S thus represents the performance degradation due to scheduling constraints. The performance degradation seen in the remaining gap, Gap P, is due to as yet unmodeled effects, such as unmodeled cache miss penalties, load imbalances, OS interrupts, and I/O, that affect actual delivered performance. The bounds are generally expressed as lower bounds on run time and, along with measured run time, are given in units of CPF (clocks per essential floating–point operation). The reciprocal of CPF times the clock rate in MHz yields an upper bound on performance in MFLOPS. The *MACS* model has been effectively demonstrated and applied in varying amounts of detail to a wide variety of processors. [1][2][3][4][5][6] The MACS model developed for each SP2 node incorporates a significant additional refinement beyond these earlier studies by including the effects of the memory hierarchy.

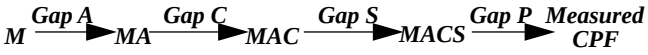


**Figure 1:** MACS Performance Bound Hierarchy

Scientific applications often exhibit a large degree of potential parallelism in their algorithms, and hence are prime candidates for execution on MPPs. Gap P can become very large and poses the fundamental limit to scalability for parallel applications. Characterizing the communication performance of an MPP and the communication requirements of an application achieves a refinement of Gap P that is crucial to modeling and improving the performance of parallel scientific applications that exhibit moderate to high amounts of communication relative to computation. We demonstrate that coupling the MACS bounds hierarchy, which models the computation of an individual node, with models of communication, as well as load balancing and cache effects, to refine Gap P, enables the effective modeling of the performance of scientific applications on a parallel computer such as the IBM SP2. This refinement is begun in this paper with the introduction of a communication model for the SP2. The MACS model is applied to the SP2 and integrated with this communication model in a case study of a commercial scientific

application.

This methodology could easily be extended to other message passing MPPs, such as the Intel Paragon and the Thinking Machines CM5. [7][8] We believe that it can also be extended to shared memory MPPs, such as the Kendall Square Research KSR2, the Convex Exemplar, and the Cray T3D [11][12][13][14], by using techniques to expose and characterize the implicit communication, as demonstrated in [6][9][10].

All experiments in this paper were run on an IBM SP2 with 32 Thin Node 66 POWER2 processors running the AIX 3.2.5 operating system. An overview of the SP2 architecture is given in Section 2. The single–node SP2 (POWER2) MACS model is detailed in Section 3, with extensions to include the effects of the memory hierarchy. Section 4 presents the communication model for the IBM SP2.

An industrial structural finite element modeling code, FEMC, is used to demonstrate the methodology. This application is being ported from a vector supercomputer, parallelized, and tuned for the IBM SP2 at the University of Michigan, and hence represents one scenario of performance modeling. Three performance–limiting FEMC routines are examined. These routines exhibit low, moderate, and high communication to computation ratios and various patterns of communication. Section 5 gives a brief overview of the application routines and discusses the performance modeling results. The performance models developed via this methodology facilitate explaining performance, identifying performance bottlenecks, and guiding application code improvements.

## 2. IBM SP2 Architectural Analysis

A typical IBM SP2 contains between 4 and 128 nodes connected by a High–Performance Switch communication interconnect; bigger configurations are possible. Each node consists of a POWER or POWER2 processor and an SP2 communication adapter. There are three types of POWER2 nodes currently available: Thin Nodes, Thin 2 Nodes, and Wide Nodes. Thin Nodes have a 64 Kbyte data cache; Thin 2 Nodes have a 128 Kbyte data cache, and Wide Nodes have a 256 Kbyte data cache. Thin 2 Nodes and Wide Nodes can do quadword data accesses, copying two adjacent double words into two adjacent floating point registers from the primary cache in one cycle.

### 2.1. POWER2 Architecture [15]

Each Thin Node POWER2 operates at a clock speed of 66.7 MHz, corresponding to a 15 ns processor clock cycle time. The POWER2 processor is subdivided into an Instruction Cache Unit (ICU), a Data Cache Unit (DCU), a Fixed–Point Unit (FXU), and a Floating–Point Unit (FPU), as shown in Figure 2. The POWER2 processor also includes a Storage Control Unit (SCU) and two I/O Units (XIO), neither of which are shown or described. Thin Nodes and Thin 2 Nodes also include an optional secondary cache (1 or 2 Mbytes, respectively), memory (64 Mbytes to 512 Mbytes), and a communication adapter. All experiments in this paper are run on Thin Nodes with no secondary cache and a 256 Mbyte memory.

The ICU can fetch up to eight instructions per clock cycle from the instruction cache (I–Cache). It can dispatch up to six instructions per cycle through the dispatch unit, two of which are reserved for branch and compare instructions. Two independent branch processors within the ICU can each resolve one branch per cycle. Most of the branch operation penalties can be masked by resolving them concurrently with FXU and FPU instruction execution. The instruction cache is 32 Kbytes with 128 byte cache lines with a one cycle access time.

The FXU decodes and executes all memory references, integer operations, and logical operations. It includes address translation, data protection, and data cache directories for load/store instructions. There are two fixed–point execution units which are fed by their own instruction decode unit and maintain a copy of the general purpose register file. Two fixed–point instructions can be executed per cycle by the two execution units. Each unit contains an adder and a logic functional unit providing addition, subtraction, and Boolean operations. One unit can also execute special operations such as cache operations and privileged operations, while the other unit can perform fixed–point multiply and divide operations.

The FPU consists of three units, a double precision arithmetic unit, a load unit, and a store unit. Each unit has dual pipelines and hence can execute up to two instructions per cycle. The peak issue rate from the FPU instruction buffers to these units is thus two double precision floating–point multiply–adds, two floating–point loads, and two floating–point stores per clock.

The DCU includes a four–way set associative multiport write–back cache. The experiments in this paper were performed on a configuration with a 64 Kbytes data cache with 64 byte cache lines. The DCU also provides several buffers for cache and direct memory access (DMA) operations as well as error detection/correction and bit steering for all data sent to and received from memory. The DCU has a one cycle access time. The miss penalty to memory is determined experimentally to be between 16 and 21 processor clock cycles.

### 2.2. Interconnect Architecture [16][17] [18]

The SP2 interconnect, termed the High–Performance Switch (HPS), is designed to minimize the average latency of message transmissions while allowing the aggregate bandwidth to scale linearly with the number of nodes. The HPS is a bidirectional

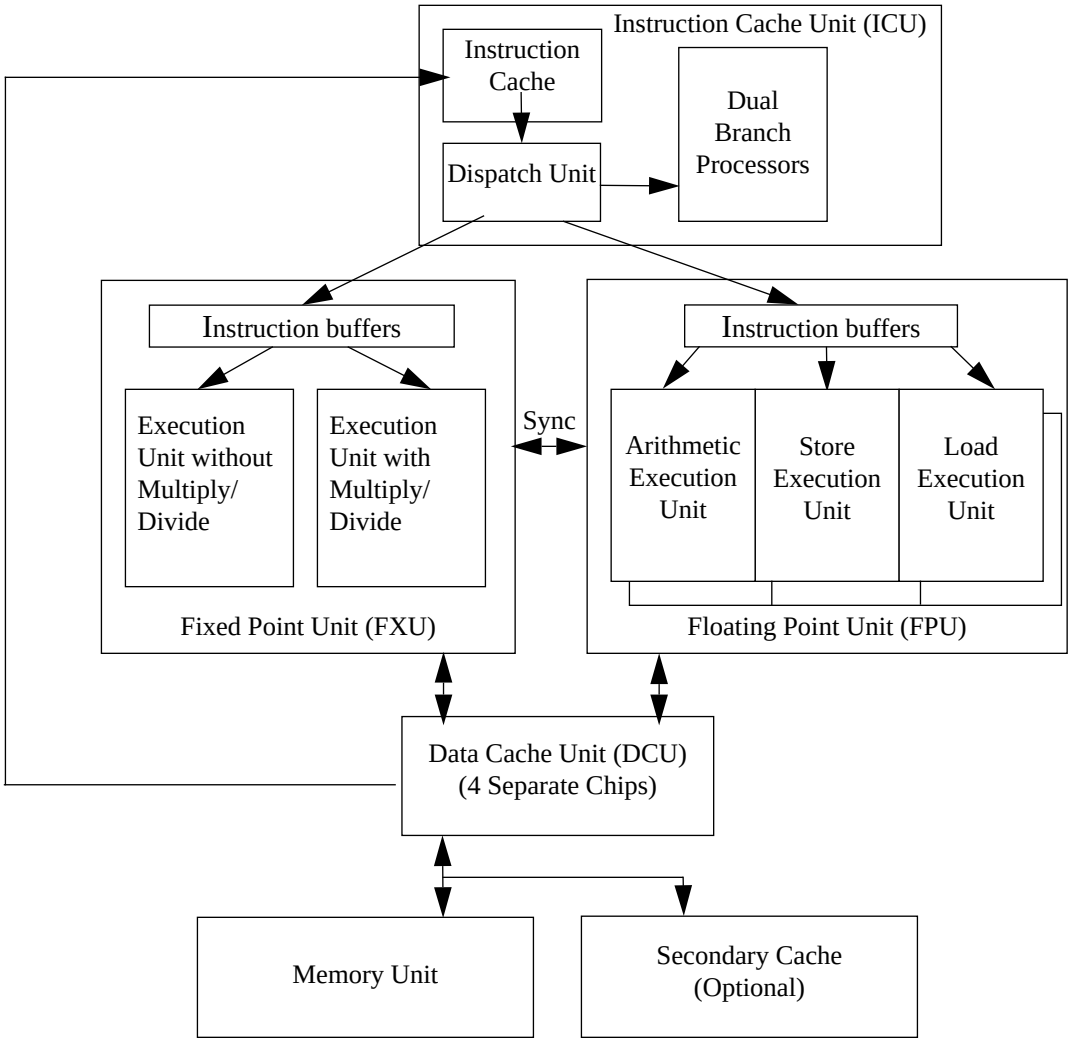


**Figure 2:** POWER2 Processor Architecture

multistage interconnect (MIN), logically composed of switching elements. It operates at 40 MHz, providing a peak bandwidth of 40 MBytes/sec in each direction over each full–duplex communication link.

Each message packet includes routing information that enables each switching element to determine on the fly the next destination for the packet. Buffered wormhole routing is employed for switch flow control. Unlike standard wormhole routing algorithms, if a message is blocked within a switching element, it is temporarily buffered in dynamically allocated shared memory. Each switching element is physically an eight input, eight output device, wired as a bidirectional 4x4 element.

Nodes of the SP2 system are grouped into frames; each frame consists of a switch board, as shown in Figure 3, and 16 nodes. Each frame incorporates 8 switching elements in two stages of four elements each, plus 8 shadow switching elements to provide a redundant check. For systems of up to eighty processors, the 16 links of a frame on the far side of the second switch stage are connected to the far side links of 1 to 4 other frames. SP2 systems with 81 to 128 nodes use intermediate switch boards between frames.

Messages to be sent between nodes on the SP2 are broken into discrete packets, each containing the required routing information. The smallest unit on which flow control is performed on the SP2, a flit, is one byte wide, corresponding to the width of an output port of the HPS switch elements. Packets vary in length up to 255 flits. Each packet is of length $r + n + 1$, where the first flit contains the packet length, the next $r$ flits contain the routing information, and the last $n$ flits contain data and error checking bytes. On a 32 node system, $r = 1$ or 2. Experiments show that the first packet can contain up to 216 bytes (or flits) of useful data. Additional packets can contain up to 232 bytes (or flits) of useful data.

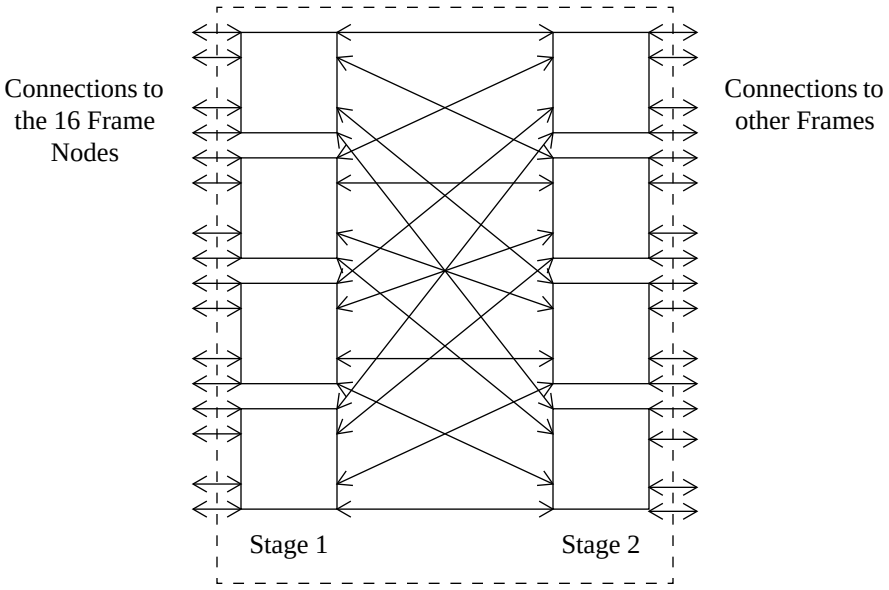


**Figure 3:** SP2 Switch Board

## 3. IBM SP2 MACS Model

A lower bound on the computation run time of floating–point applications on the IBM SP2 is given by the MACS model developed for the POWER2 processor. Only the M bound, MA bound, and MAC bounds are presented below. The MACS bound is not developed because the POWER2 architecture is implemented with extensive buffering, multiple issue and out–of–order execution among its functional pipes; hence the schedule is modified dynamically at run time and schedule stalls are greatly reduced. Thus, a realistic MACS bound is hard to develop; and would likely be very close to the MAC bound.

### 3.1. M Bound Equations

The M bound models the peak performance of the POWER2 FPU, independent of the application requirements, the compiler workload, or the scheduler. The M Bound for the POWER2 is 0.25 CPF since the POWER2 can compute at most four floating–point operations per cycle (one multiply–add issued to each of 2 pipelines).

### 3.2. MA Bound Equations

The MA bound models the peak application performance of the POWER2 architecture given the visible workload of the high level application code. The POWER2 is modeled as five independent functional units: floating–point unit, fixed–point unit, instruction issue unit, memory unit, and a dependence pseudo–unit. The MA bound, $t_{MA}$, is calculated as the maximum CPF bound among the five independent functional units:

$$t_{MA} = MAX\ (t_{fl}, t_{fx}, t_m, t_i, t_d) / (f_a + f_m + 2 * f_{ma} + 4 * f_{div} + 4 * f_{sqrt}) \quad (1)$$

The CPF bound for each functional unit is calculated as a function of the number of essential operations that must be performed by that functional unit per simulation time step in the high level source code. The bound assumes that each functional unit need execute only these essential operations, and that it can execute them at its peak rate. The compiler is ‘idealized’ in that no nonessential operations are considered; the scheduler is ‘idealized’ in that no stalls due to resource constraints and/or schedule dependences are considered. Equation (1) bounds run time by assuming that the busiest functional unit is kept continuously busy.

The set of essential arithmetic operations includes the minimum number of floating–point assembly operations necessary to complete a computation (including floating–point additions and subtractions, $f_a$, floating–point multiplications, $f_m$, triad operations which do both, $f_{ma}$, division operations, $f_{div}$, and square root operations, $f_{sqrt}$). Both division and square root operations are weighted by a factor of four, as is commonly done for the Lawrence Livermore Fortran Kernels [19] and other benchmarks. The number of essential floating–point loads, $l_{fl}$, equals the number of distinct values that appear on the right hand side (RHS) of a high level code statement before they appear on the left hand side (LHS) of a high level code statement. The number of essential floating–point stores, $s_{fl}$, equals the number of distinct values that appear on the LHS of a high level code statement

that are neither temporary values nor scalars which should spend their lifetime in registers. For Thin 2 Nodes or Wide Nodes, $l_{fl}$ and $s_{fl}$ should be divided by two if the stride equals one, since it is possible to employ quadword loads and stores.

The floating–point functional unit bound, $t_{fl}$, models the time needed in the FPU to execute the essential arithmetic and memory floating–point instructions. Since the POWER2 contains three dual–pipelined execution units in the FPU – one for arithmetic operations, one for loading data, and one for normalizing store data:

$$t_{fl} = \text{MAX}\left((f_a + f_m + f_{ma} + 17 * f_{div} + 27 * f_{sqrt}) / 2,\ s_{fl} / 2,\ l_{fl} / 2\right) \quad (2)$$

Floating–point divide operations require the FPU for 17 cycles and square root operations require 27.

The model's instruction issue functional unit bound, $t_i$, models the IBM SP2 instruction dispatch bandwidth of the ICU. This unit can dispatch four floating–point arithmetic or memory reference instructions per cycle. The POWER2 can also dispatch branch and condition register instructions concurrently with the above, or other fixed-point instructions in place of memory operations; however, since in scientific loop-dominated code they have negligible or no impact on POWER2 performance, these instructions are not included in the MA bound.

$$t_i = (f_a + f_m + f_{ma} + f_{div} + f_{sqrt} + l_{fl} + s_{fl}) / 4 \quad (3)$$

Since non-floating–point operations are assumed to have a negligible impact on the performance of scientific applications, the fixed–point functional unit, $t_{fx}$, models only the impact of the fixed–point unit on floating–point operations. An address calculation is required for each floating–point memory operation. Since the FXU in the POWER2 architecture can begin two floating–point memory operations (either loads or stores) per cycle:

$$t_{fx} = (l_{fl} + s_{fl}) / 2 \quad (4)$$

In previously published implementations of the MACS model, the memory hierarchy unit, $t_m$, has ignored the instruction cache entirely, and has assumed that all data accesses hit in the data cache. As a result, the memory unit models developed for other architectures have focused on data cache port bottlenecks only. Since the routines of interest modeled in Section 5 have working sets that exceed the size of the data cache, a more sophisticated model of memory is required to explain a significant fraction of runtime. Two ports connect the floating–point unit with the primary data cache in the POWER2 processor, and a single port connects the data cache to memory. A memory hierarchy unit lower bound on run time is as follows:

$$t_m = \text{MAX}((\text{load miss time} + \text{store miss time}),\ (l_{fl}\, L_{eff} + s_{fl}\, S_{eff}) / 2) \quad (5)$$

The first term in Equation (5) models the single port between the data cache and memory. In applications with a high miss rate, multiple cache misses may occur at the same time, but only one can be serviced by the memory at time due to this single port bottleneck.

The second term in Equation (5) models the two ports between the FPU and the data cache. In applications with a low miss rate, there is typically only a single outstanding cache miss at any one time. While the memory services the miss, the other data cache port can continue to service a single memory access per cycle. The effective number of cycles per floating–point load, $L_{eff}$, and per floating–point store, $S_{eff}$, is calculated as a function of the number of essential misses. Note that $t_m \geq t_{fx}$, and $t_m = t_{fx}$ only if all memory access operations hit in the data cache.

A key issue in evaluating Equation (5) is developing a lower bound on the effective access time of floating–point loads and stores, given a high level application. For a memory system of $n$ levels, the Effective Access Time, $T_{eff}$, is:

$$T_{eff} = \sum_{j=1}^{n} f_j \cdot t_j \quad (6)$$

where $f_j$ is the Access Frequency for level $j$, and $t_j$ is the Access Time for that level. The access frequency can be calculated as follows:

$$f_j = m_1\, m_2 \ldots m_{j-1}\, (1\text{-}m_j) \quad (7)$$

where $m_j$ is the miss ratio in the $j$th level of the memory hierarchy. SP2 systems have a primary cache, an optional secondary cache, and main memory, hence Equation (6) can be rewritten:

$$T_{eff} = (1 - m_1)t_1 + m_1\,(1 - m_2)\,t_2 + m_1\, m_2\,(1\text{-}m_3)\,t_3 = (1 - m_1)\,t_1 + m_1\,t_3 \quad (8)$$

assuming no secondary cache ($m_2 = 1$) and ignoring page faults ($m_3 = 0$). Experimental calibration loops show that $t_{1l} = t_{1s} = 1$, $t_{3l} = 16.3$, and $t_{3s} = 22.0$, hence:

$$L_{eff} = 1 + 15.3 * m_{1l} \qquad S_{eff} = 1 + 21.0\, m_{1s} \quad (9)$$

Thus for our SP2 system, and later experiments, we use:

$$t_m = \text{MAX}(((l_{fl}\,(16.3\, m_{l1}) + s_{fl}\,(22.0\, m_{s1})),\ ((l_{fl}\,(1 + 15.3\, m_{l1}) + s_{fl}\,(1 + 21.0\, m_{s1})) / 2) \quad (10)$$

Essential misses are classified as either compulsory ($m_{comp}$) or capacity ($m_{cap}$). Hence a lower bound on the miss rate, $m$, can be calculated as follows:

$$m = \frac{m_{comp} + m_{cap}}{\text{Number of Essential Accesses}} \quad (11)$$

A lower bound on the number of compulsory misses equals the number of blocks in the Working Set ($B$), hence:

$$m_{comp} = B \quad (12)$$

Capacity misses depend on the number of Working Set blocks ($B$), the number of Cache blocks ($C$), the Cache Degree of Associativity ($A$), and the Access Pattern. Assuming a linear access pattern, as is often found in loops, a lower bound on capacity misses, $m_{cap}$, can be calculated as follows:

$$m_{cap} = \begin{cases} 0 & B \le C \quad \text{(cache region)} \\ B\dfrac{B-C}{(DC)/A} & C \le B \le C\,(1 + D/A) \quad \text{(transition region)} \\ B & B \ge C\,(1 + D/A) \quad \text{(memory region)} \end{cases} \quad (13)$$

$D$ is the number of degrees of freedom in the access pattern which equals the number of distinct arrays and is restricted to be between 1 and $A$. The miss ratio is linear in the transition region [2], and $m_{cap}$ equals the product of $B$ and the miss ratio.

The loop–carried dependence pseudo–unit, $t_d$, models the performance of loops with a recurrence, i.e. a result of one iteration depends on the corresponding result of a previous iteration. Whenever there is such a cycle in the dependence graph of the floating–point arithmetic operations, $t_d$ is computed as the worst- case recurrence cycle, where each recurrence cycle is calculated as the total latency of the operations in one tour divided by the number of iterations in that recurrence cycle. The latency of an operation is related to pipeline depth and is computed as the minimum number of clocks between issuing that operation and issuing a succeeding operation that uses its result as an operand, hence:

$$L_r = \text{total latency of the loop-carried dependence in recurrence cycle } r \quad (14)$$

$$I_r = \text{number of iterations in recurrence cycle } r \quad (15)$$

$$t_d = MAX_{\forall\ recurrence\ cycles\ r}(L_r/I_r) \quad (16)$$

The latency is 1 cycle for floating–point adds and multiplies, 2 for floating–point multiply adds, 17 for floating–point divides, and 27 for floating–point square roots.

### 3.3. MAC Bound Equations

The MAC bound for the POWER2 architecture is computed similarly to the MA bound, except that the operation counts are computed from the compiled assembly code. In the MAC bound equations, counts of the various types of operations are marked with primes to indicate that they represent the number of operations found in the compiled code, not the minimum number of essential operations needed in the high–level code.

Furthermore all compiled operations are counted in the MAC model, including fixed–point and branch instructions. The fixed–point unit can begin two fixed–point operations per cycle, but at most one fixed–point multiplication or division per cycle. The branch functional unit, which can execute two branch instructions per cycle, is added for the MAC model. $L_{eff}'$ and $S_{eff}'$ are calculated as for the MA model, but the miss ratio is computed using the number of actual misses divided by the actual accesses. For our SP2, $t_m'$ is calculated as in Equation (10), using the MAC parameter values.

$$n_{FPU} = \text{number of FPU computation instructions} = f_a' + f_m' + f_{ma}' + 17 * f_{div}' + 27 * f_{sqrt}' + \text{others} \quad (17)$$

$$t_{fl}' = MAX\ (n_{FPU} / 2,\ l_{fl}' / 2,\ s_{fl}' / 2) \quad (18)$$

$$n_{FXU} = \text{number of FXU instructions} = l_{fl}' + s_{fl}' + \text{others} \quad (19)$$

$$n_{FXMD} = \text{number of fixed–point multiplication and division instructions} \quad (20)$$

$$n_{BC} = \text{number of branch and compare instructions} \quad (21)$$

$$t_{fx}' = MAX\ (n_{FXU} / 2,\ n_{FXMD}) \quad (22)$$

$$t_m' = MAX((\text{load miss time} + \text{store miss time}),\ ((l_{fx}' + l_{fl}')\ L_{eff}' + (s_{fx}' + s_{fl}')\ S_{eff}') / 2) \quad (23)$$

$$t_b' = n_{BC} / 2 \quad (24)$$

$$t_i' = (\text{compiled code length} - n_{BC}) / 4 \quad (25)$$

$$t_d' = MAX_{\forall\ recurrence\ cycles\ r}(L_r'/I_r) \quad (26)$$

$$t_{MAC} = MAX\ (t_{fl}', t_{fx}', t_m', t_b', t_i', t_d') / (f_a + f_m + 2 * f_{ma} + 4 * f_{div} + 4 * f_{sqrt}) \quad (27)$$

### 3.4. Automatic MAC Bound Generator

The Automatic MAC Bound Generator is a single pass forward-scanning tool which generates the parameters used in the

MAC Bound. [6] It reads a designated region of interest of the IBM POWER2 assembly code as its input and reports statistics for each loop. Reported statistics include the nesting relationship for each loop and respective values of $t_{fl}'$, $t_{fx}'$, $t_b'$, and $t_i'$. This tool accepts perfectly nested loops and most imperfectly nested loops (forward branches inside the loop body are allowed) as input. In general, all nested loops written in well-structured programming styles are accepted by this tool. Statistics for outer loops report only on code not contained in the inner loops, i.e. the *residue* code; statistics for code spanned by forward branches are reported separately. Forward branches within loops often complicate the computation of the bounds; however, the tool can recognize and handle such branches appropriately using weights derived from profiling. These statistics are then combined in a weighted average, with the weights determined by standard basic block profiling.

## 4. IBM SP2 Communication Model

Scientific applications executed on MPPs tend to exhibit a large Gap P due to communication overhead, nonessential cache stalls, load imbalance, other unmodeled effects, and system level phenomena. Extending the parameter–based hierarchical performance bounds modeling methodology into Gap P can begin with developing performance models for communication as a function of simple performance parameters. One such estimation approach has been applied to a wide variety of parallel machines with good results. Simple calibration loops were developed to measure latency and bandwidth for internode communication. [20] Let $r_\infty$ be the asymptotic transfer rate of a communication interconnect in units of megabytes/second, $n$ is the message length in bytes, and $t_o$ is the (asymptotic) zero message length latency in microseconds. This suggests the following model of communication latency:

$$T_{comm}(n) = t_o + \frac{n}{r_\infty} \tag{28}$$

As shown in Section 4.1, Equation (28) works well in characterizing point–to–point communication.

For more complex communication patterns such as one–to–many, many–to–one, many–to–many, combine, and broadcast, both $t_o$ and $r_\infty$ are found to be functions of the number of processors, $p$. We define $t_{comm}(p)$ as the setup time and $\pi_{comm}(p)$ as the transfer time per byte. This suggests the following model of communication latency:

$$T_{comm}(n, p) = t_{comm}(p) + \pi_{comm}(p) \bullet n \tag{29}$$

In the rest of Section 4, we model the performance of the IBM SP2 in the form of Equation (29) for a variety of communication constructs defined in the message passing library, MPL. Although these models are actually estimates derived from curve–fitting to clean machine primitive test loop performance, we have found their predictive accuracy to be good enough to permit treating them as performance bounds. Furthermore there is no overlap in our case study between the computation modeled in the MAC bound and the communication since all the communication patterns are blocking. As of this date, our experiments show that our SP2 always exhibits blocking behavior even for commands specified as nonblocking. Thus the communication model can simply be added to the MAC bound.

### 4.1. Point–to–Point Communication

The cost of point–to–point communication is measured by a classic “ping–pong” experiment. Processor P1 executes a blocking send, MP_BSEND, to processor P2, and then executes a blocking receive, MP_BRECV, from P2. Meanwhile P2 executes a blocking receive, MP_BRECV, from P1, and then executes a blocking send, MP_BSEND, to P1. Both processors loop for many iterations, and the minimum time is divided by 2 to determine the latency for a single point–to–point communication on an otherwise clean system. As shown in Figure 4, Equation (28) applies well if the experimental results are split into four distinct regions as a function of the message length. The resulting values for $t_o$, and $r_\infty$ are shown in Table 1.

The length of a packet is at most 255 bytes, including the actual data, routing information, packet length information, and error checking bits. This suggests why there is a change in parameter values at $n$ = 217. The length of the FIFO queue is 2 Kbytes, suggesting why they change again at $n$ = 2048. The change in parameter values at $n$ = 32 Kbytes is due to the fact that MPL does one copy of data for messages of size greater than 32 Kbytes, and two copies of data for messages of size less than or equal to 32 Kbytes. In successive cases, the asymptotic zero message length latency increases, but this effect is immediately compensated for by the change in the transfer time. As a result, after each transition point, the point–to–point communication time as a function of the message length is lower than would have been expected from the model(s) of smaller messages.

Additional experiments indicate that the difference between intra–frame (nearby node) and inter–frame (remote node) communication latencies is approximately 1 μs, independent of message length; this is less than 3% of the total latency for the smallest messages. This effect is negligible for larger messages.

### 4.2. One–to–Many Communication

The cost of a scatter operation, implemented as a one–to–many communication, is measured by having one processor send distinct messages to $P$ other processors by executing a non–blocking send, MP_SEND, to each in turn. At the end of every it-

r

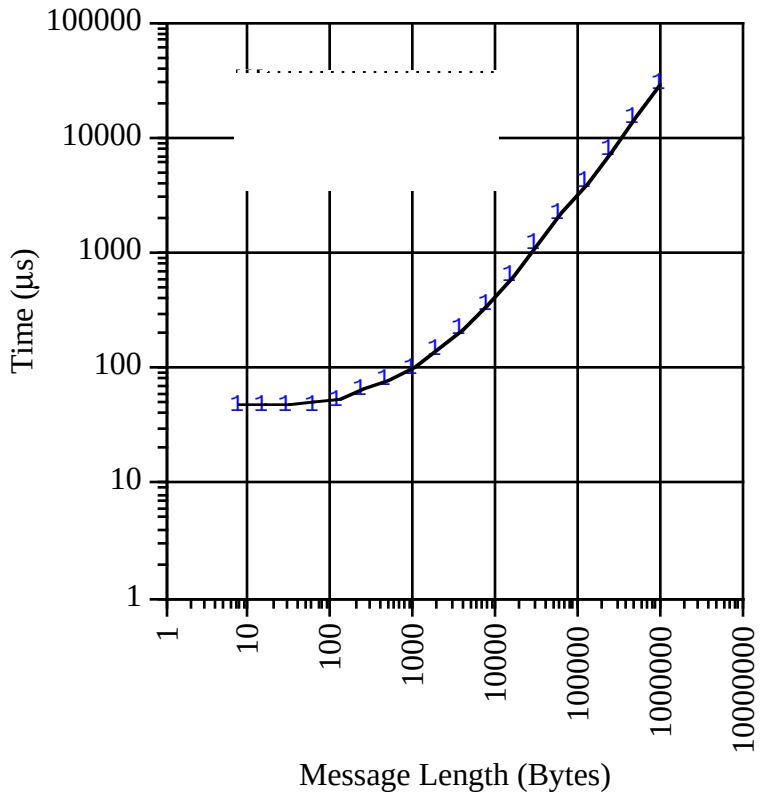


**Figure 4:** Point–to–Point Communication Time

| $n$ | $t_o$ | $r_\infty$ |
|---|---|---|
| $n < 217$ | 47 | 23.5 |
| $217 \le n \le 2048$ | 55 | 22.6 |
| $2048 < n \le 32768$ | 74 | 29.3 |
| $32768 < n$ | 399 | 36.2 |

**Table 1:** Point–to–Point Communication Parameters

eration MP_WAIT is called to ensure that all of the sends have completed. This series of operations is done for a large number of iterations; the total time is measured and averaged for a single iteration. Each receiving processor executes a blocking receive, MP_BRECV, per iteration. As shown in Figure 5, Equation (29) applies fairly well over a range of $p$ and $n$ values if $t_{comm} = -5.5 + 15.5 * p$ and $\pi_{comm} = 0.031\ p$. As a result:

(30)

The setup time is 10.0 µs for a single destination, and 15.5 µs for each additional destination, apparently due to the added overhead of managing more than one ongoing message. As $n$ increases toward 64 Kbytes, the overall transfer rate goes to 32 MB/s.

### 4.3. Many–to–One Communication

The cost of a gather operation from $P$ processors to one, implemented as a many–to–one communication, is measured by having $P$ processors execute a blocking send, MP_BSEND, to one other processor. This series of operations is done for a large number of iterations. The receiving processor executes a non–blocking receive, MP_RECV, for each source every iteration. At the end of every iteration MP_WAIT is called to ensure that all of the sends have completed. The total time is measured and averaged for a single iteration. As shown in Figure 6, Equation (29) applies fairly well to the range of $p$ and $n$ values if $t_{comm} = 3.0 + 13.3 * p$ and $\pi_{comm} = 0.0285\ p$. As a result:

(31)

As with a one–to–many communication, as the message length $n$ goes towards 64 Kbytes, the overall transfer rate approaches

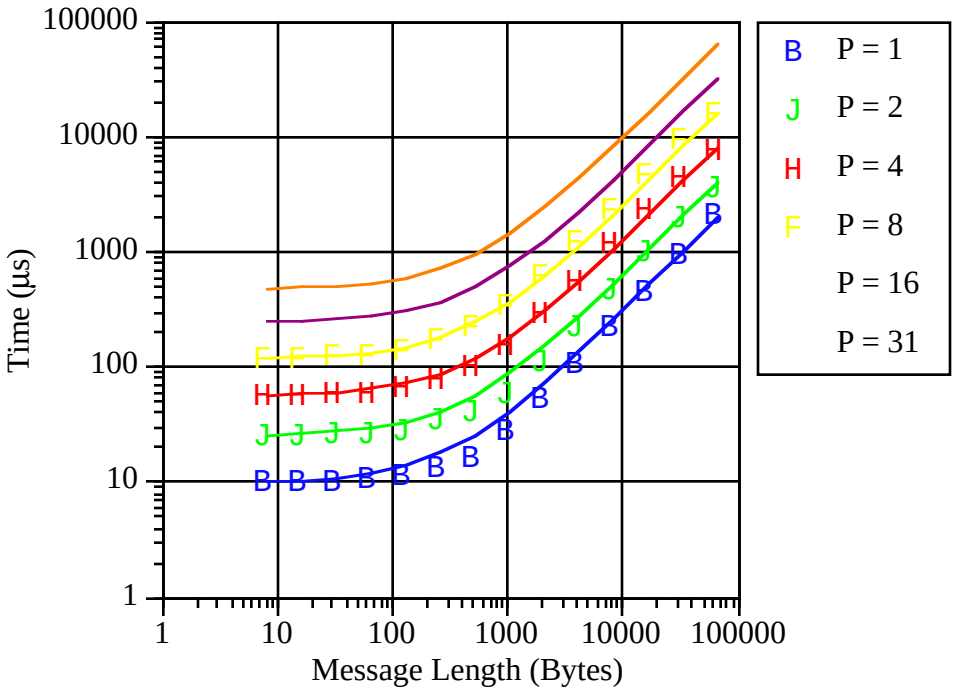


**Figure 5:** One–to–Many Communication Time

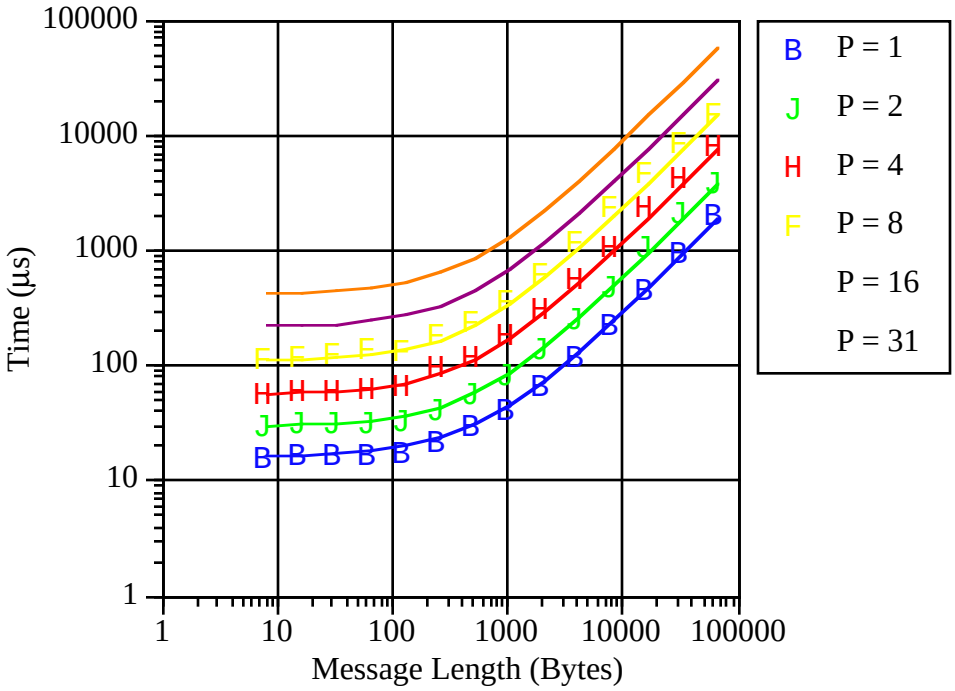


**Figure 6:** Many–to–One Communication Time

35 MB/s.

### 4.4. Many–to–Many Communication

The cost of a many–to–many communication is measured by having each of P processors send a distinct message to each

of the other processors. Each processor executes a non–blocking send, MP_SEND, to every other processor. Each processor then executes a non–blocking receive, MP_RECV, from every other processor. At the end of every iteration MP_WAIT is called to ensure that all of the sends have completed. This series of operations is done for a large number of iterations. The total time is measured and averaged for a single iteration. As shown in Figure 7, Equation (29) applies to a range of $p$ and $n$ values if $t_{comm} = 43 + 40\ (p-2)$ and $\pi_{comm} = 0.057 + 0.062\ (p-2)$. As a result:

$$T_{MM}(n,p) = (43 + 40 \cdot (p-2)) + (0.057 + 0.062 \cdot (p-2)) \cdot n \qquad (32)$$

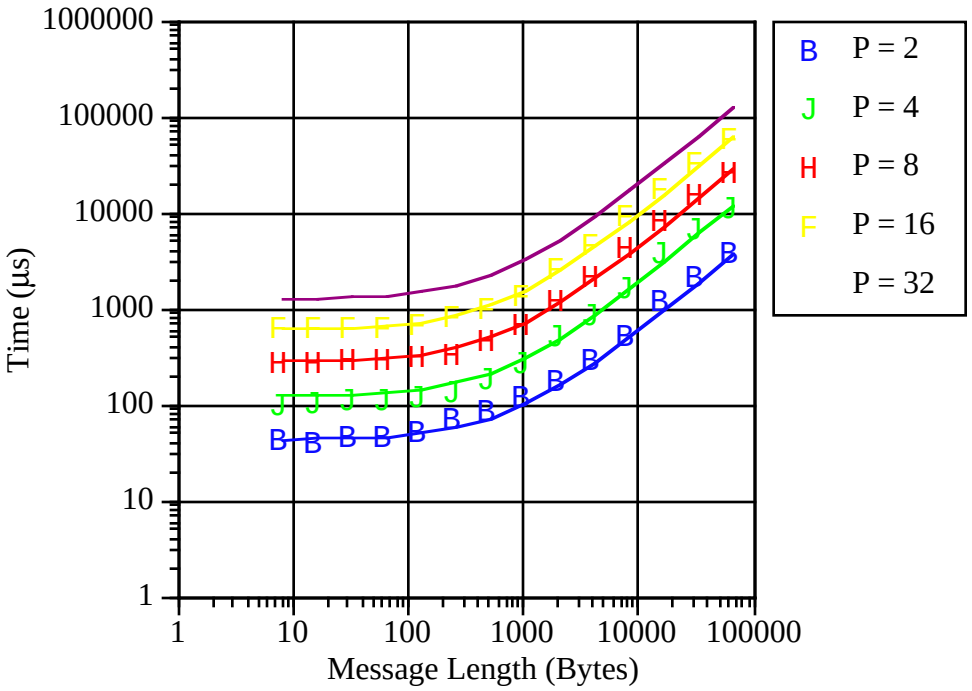


**Figure 7:** Many–to–Many Communication Time

For large messages, the bidirectional rate for a processor node ≈ 32 MB/s, implying good communication scalability even when the HPS is highly saturated.

### 4.5. Combine Communication

The cost of a combine operation, implemented using an MPL collective communication routine, is measured by having each of P processors execute MP_COMBINE. This results in a vector of double precision operands being sent to one processor. The receiving processor performs a double precision addition reduction and broadcasts the results back to the P processors. This operation is done for a large number of iterations and the total time is measured and averaged for a single iteration. As shown in Figure 8, Equation (29) applies to a range of $p$ and $n$ values as follows:

$$T_{CB}(n,p) = \begin{cases} 97D + (0.11D)\,n & n < 217 \\ 114D + (0.12D)\,n & 217 \le n \le 2048 \\ (-50 + 191D) + (0.09D)\,n & 2048 < n \end{cases} \qquad (33)$$

$$\text{where} \quad D = \lfloor lg_2(p) \rfloor$$

### 4.6. Broadcast Communication

The cost of a broadcast operation, implemented using an MPL collective communication routine, is measured by having every processor execute a broadcast, MP_BCAST, specifying the same source, P0. This series of operations is done for a large number of iterations. The total time is measured for P0, and averaged for a single iteration. As shown in Figure 9, Equation (29) applies to a range of $p$ and $n$ values as follows:

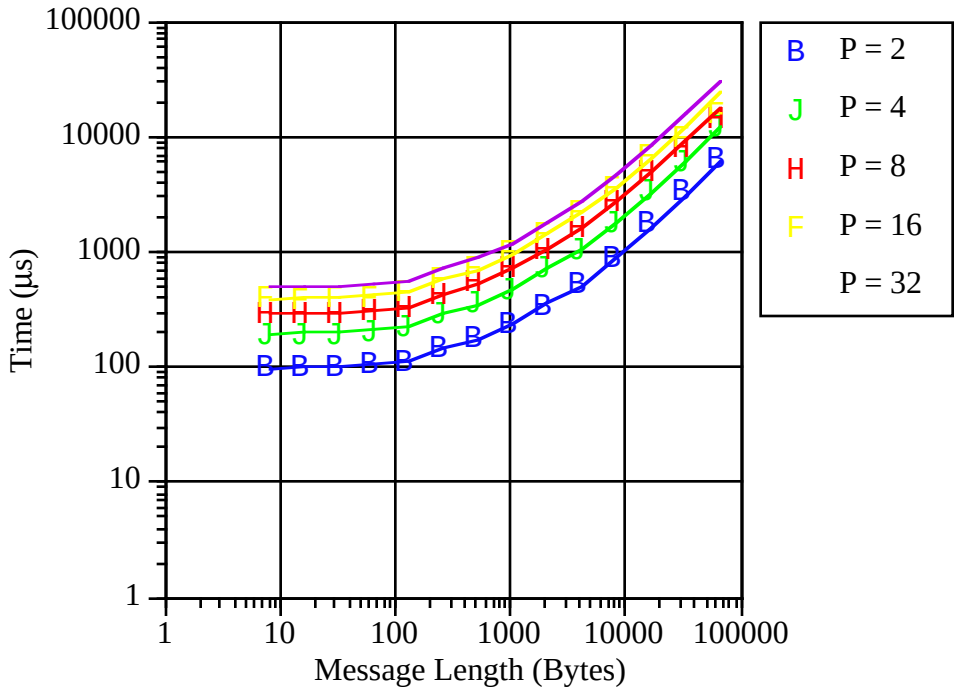


**Figure 8:** Combine Communication Time

$$T_{BC}(n, p) = \begin{cases} (9.6 + 14 \cdot D) + (0.0083 + 0.015 \cdot D)\, n & n < 217 \\ (6.0 + 12 \cdot D) + (0.0250 + 0.026 \cdot D)\, n & n \geq 217 \end{cases} \quad (34)$$

where $D = \lfloor lg_2(p) \rfloor$

Broadcasting to P processors takes about the same time as broadcasting to the nearest lowest integer power of 2 processors, suggesting that broadcast is done using a recursive doubling algorithm.

## 5. Performance Evaluation and Improvement of FEMC

This section analyzes the performance of three important routines drawn from FEMC, an industrial finite element code. All the modeling data shown is collected from a single (typical) simulation time step. The calculated miss ratio for the MA bound assumes cache lines are accessed with a stride of one, which serves as a lower bound. The MAC bounds were calculated using the miss ratios calculated for the MA bound. (PLEASE NOTE: Although this can be used as a lower bound, for the final draft of the paper we will use the actual cache miss counts using the POWER2 Performance Monitor. [21]) An additional bound, MA_PC, is also shown since $t_m$ dominates the MA and MAC bounds for all of the modeled routines and it is interesting to examine the bottlenecks masked by the memory component of the bound. This MA_PC bound is the MA bound with a perfect cache (hit ratio of 100%) assumption.

### 5.1. Routine A

Routine A exhibits the longest execution time of the three routines. It is characterized by large amounts of computation and communication. The communication pattern is many–to–many and is nonuniformly distributed throughout the routine. It employs MP_SEND and MP_BRECV communication primitives.

Figure 10 shows the measured run time and modeled behavior of Routine A for each processor in an eight processor run. While the modeled behavior can be seen to exhibit very good load balancing, particularly with regard to computation, the measured performance exhibits somewhat more unbalanced, although acceptable, behavior. The large unmodeled gap between the MAC + Communication model and the measured time can be attributed to unmodeled data cache misses, unmodeled aspects of communication (e.g. nonzero processor wait times during serial code sections and load imbalances), poor instruction overlap, and interrupts.

Figure 11 shows the measured run time of P0 in Routine A multiplied by the number of processors, *P*. Hence perfect speedup would yield a constant bar height, independent of *P*. As can be seen, although there is significant speedup, there is some

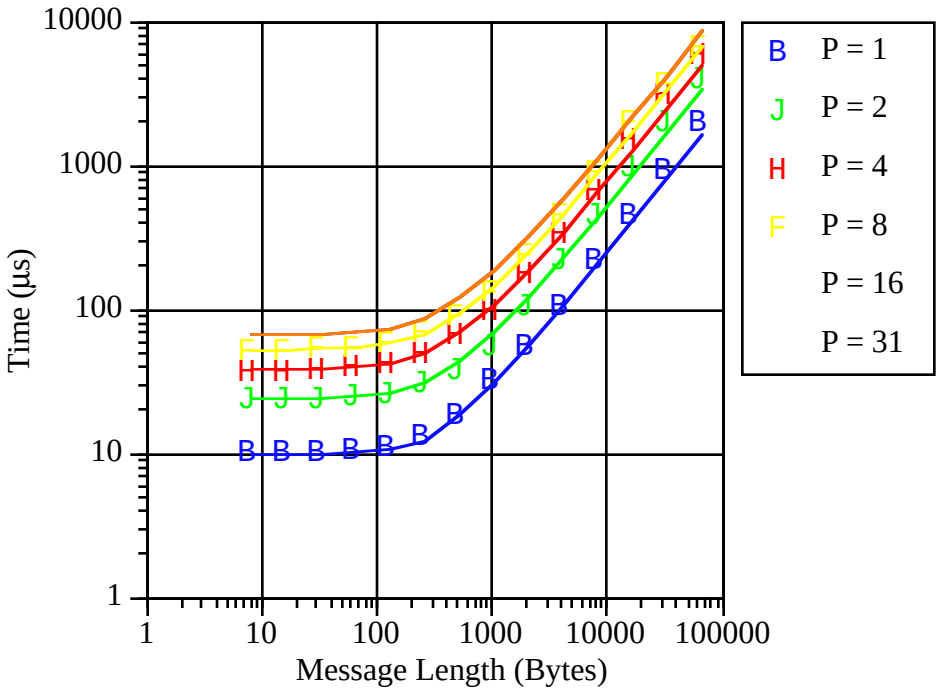


**Figure 9:** Broadcast Communication Time

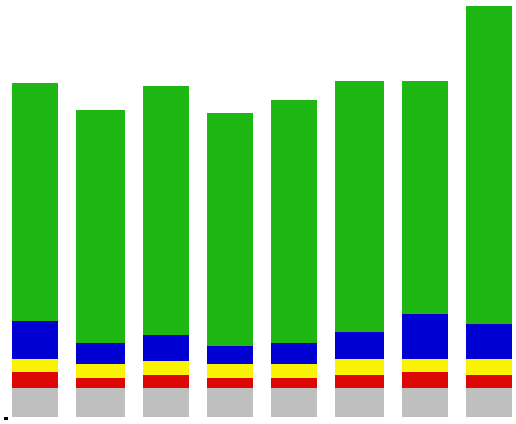

**Figure 10:** Load Balancing in Routine *A* with 8 Processors

overhead, which is only partially attributable to the modeled communication overhead. The modeled run time explained by MA_PC is essentially unchanged (except for the additional loop overhead introduced by parallelization) as the number of processors is increased. The modeled run time explained by the MA bound varies significantly with the number of processors. As P increases, the working set size decreases, allowing it to fit in cache when P = 4. (PLEASE NOTE: The MAC bound tracks the MA bound because, in this draft of the paper, it uses the miss ratio calculated for the MA bound. This will be fixed in the

final draft of the paper as noted above.) As the number of processors is increased, the number of MP_SEND and MP_BRECV communication primitives scales proportionally, causing $T_{comm}$ to increase as shown.

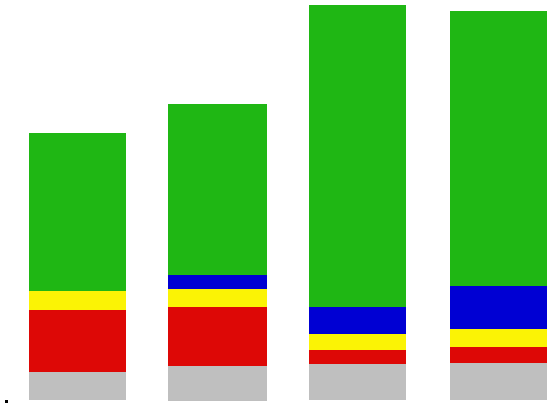

**Figure 11:** (# Processors * Time) in Routine *A*, P0

### 5.2. Routine B

Routine B exhibits the shortest execution time of the three routines. It is communication–intensive with relatively little calculation. The communication pattern is a gather (many–to–one) followed by broadcast (one–to–many with identical data). It is implemented using MP_RECV, MP_BSEND, and MP_BCAST. Computation in this routine begins only after the communication is completed.

Figure 12 shows the measured run time and modeled behavior of Routine B for each processor in an eight processor run. In this communication–dominated routine, the MA and MAC models are insignificant. While the measured behavior is seen to exhibit reasonable load balancing, the modeled communication is distinctly bimodal. The extra modeled communication time for P0 is due to the fact that P0 receives a many–to–one communication burst from all the other processors. P0 then performs some computation while the other processors stall, an effect which is unmodeled for P1 through P7. Only after P0 completes its computation does it broadcast to the other processors. The unmodeled gap between the MAC + Communication model for P0 and the measured time can be attributed to the communication load imbalance and OS interrupts.

Figure 13 shows the measured run time of P0 in Routine B (presented as in Figure 11). Comparing the measured and modeled run time of a particular processor, P0, as the total number of processors is increased shows a wide variation in the percentage of run time explained by communication. With only one processor, $T_{comm}$ explains only 13% of run time. As the routine is written, P0 will send a message to itself on a one processor run. As *P* is increased to 2, 4, and 8, $T_{comm}$ explains 54%, 65%, and 46% of P0's run time, respectively. As can be seen, there is very little speedup in Routine B as there is very little computation and P0 is a bottleneck during the communication phase.

### 5.3. Routine C

Routine C is calculation–intensive with relatively little communication. There are two separate global reductions (combining trees) implemented with MP_COMBINE, one in the middle of the routine, and one at the end.

Figure 14 shows the measured run time and modeled behavior of Routine C for each processor on an eight processor run. The modeled behavior can be seen to exhibit very good load balancing, both with regard to computation and communication, as does the measured performance. The unmodeled gap between the MAC + Communication model and the measured time can be attributed primarily to the unmodeled data cache misses.

Figure 15 shows the measured run time of P0 in Routine C (presented as in Figure 11). Comparing the measured and mod-

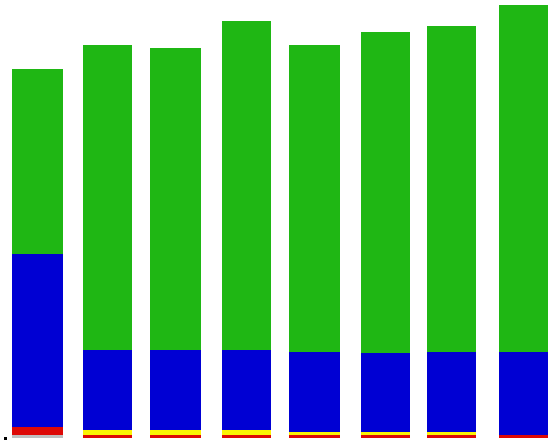

**Figure 12:** Load Balancing in Routine *B* with 8 Processors

**Figure 13:** (# Processors * Time) in Routine *B*, P0

eled run time of a particular processor, P0, as the total number of processors is increased shows that the run time explained by the MA_PC bound stays constant, and hence the amount of essential computation scales perfectly. The run time explained by the MA bound decreases as more of the working set fits in cache.

The extra overhead in the MAC bound is due to the method of loop parallelization employed in the high level code. Each processor loops over the entire iteration space. Inside the loop, each processor checks a conditional to determine if the data utilized in that iteration is local. The overhead of checking the conditional remains constant in every processor, regardless of the

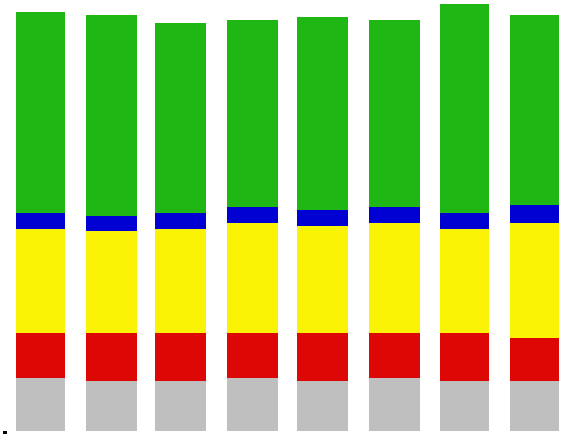

**Figure 14:** Load Balancing in Routine *C* with 8 Processors

number of processors. The MAC bound summed over all processors thus increases significantly with the number of processors. Since checking the conditional statement is not essential, this effect does not occur in the MA bound. Note that Figure 15 serves to highlight the effect of this style of coding which can easily be removed.

As the number of processors increases, the width of the MP_COMBINE increases. $T_{comm}$ increases with the log base 2 of the number of processors, but is relatively insignificant compared to the time spent in computation. Although there is significant speedup, there is also some overhead, which is only slightly attributable to the modeled communication overhead.

**Figure 15:** (# Processors * Time) in Routine *C*, P0

## 6. Conclusion

We have shown that computation and communication can be effectively bounded on the IBM SP2 using the MACS Hierarchal Performance Bounds and the SP2 communication model developed in this paper. We have demonstrated the use of this methodology to provide an intuitive understanding of where time is spent within three time–critical routines from a commercial code. While communication and computation are important parts of the performance puzzle, effects as yet unmodeled including load imbalance and nonessential cache misses also contribute significantly to performance degradation. Use of such modeling techniques leads directly to better understanding of new architectures and application codes and to their improvement.

## Acknowledgments

The University of Michigan Center for Parallel Computing, site of the IBM SP-2, is partially funded by NSF grant CDA-92-14296. We are most grateful to Doug Ranz of IBM for his sound advice and help in procuring salient facts about the IBM SP2 architecture.

## References

[1] E. L. Boyd, E. S. Davidson. "Hierarchical Performance Modeling with MACS: A Case Study of the Convex C-240," *Proceedings of the 20th International Symposium on Computer Architecture*, May, 1993, pp. 203-212.

[2] W. H. Mangione–Smith, T–P. Shih, S. G. Abraham, E. S. Davidson, "Approaching a Machine–Application Bound in Delivered Performance on Scientific Code," *IEEE Proceedings*, August, 1993, pp. 1166-1178.

[3] W. H. Mangione–Smith, S. G. Abraham, E. S. Davidson, "A Performance Comparison of the IBM RS/6000 and the Astronautics ZS–1," *Computer*, January, 1991, pp. 39-46.

[4] P. H. Wang, "Hierarchical Performance Modeling with Cache Effects: A Case Study of the DEC Alpha," University of Michigan, Technical Report, CSE-TR-232-95, March, 1995.

[5] W. Azeem. "Modeling and Approaching the Deliverable Performance Capability of the KSR1 Processor," University of Michigan, Technical Report, CSE-TR-164-93, June, 1993.

[6] E. L. Boyd, W. Azeem, H–H. Lee, T–P. Shih, S–H. Hung, E. S. Davidson. "A Hierarchical Approach to Modeling and Improving the Performance of Scientific Applications on the KSR1," *Proceedings of the 1994 International Conference on Parallel Processing*, August, 1994, Vol. 3, pp. 188-192.

[7] *Paragon XP/S Product Overview*, Supercomputer Systems Division, Intel Corporation, Beaverton, OR, 1991.

[8] *Connection Machine CM5 Technical Summary*, Thinking Machines Corporation, Cambridge, MA, November, 1992.

[9] E. L. Boyd, J. D. Wellman, S. G. Abraham, E. S. Davidson. "Evaluating the Communication Performance of MPPs Using Synthetic Sparse Matrix Multiplication Workloads," *Proceedings of the 1993 International Conference on Supercomputing*, July, 1993, pp. 240-250.

[10] E. L. Boyd, E. S. Davidson. "Communication in the KSR1 MPP: Performance Evaluation Using Synthetic Workload Experiments," *Proceedings of the 1994 International Conference on Supercomputing*, July, 1994, pp. 166-175.

[11] *KSR1 Principles of Operation*, Kendall Square Research Corporation, Waltham, MA, 1992.

[12] *KSR1 Technical Summary*, Kendall Square Research Corporation, Waltham, MA, 1992.

[13] *Convex Exemplar Programming Guide*, Convex Press, Richardson, Texas, 1994.

[14] *Cray T3D System Architecture Overview*, Cray Research, Inc., Chippewa Falls, WI, September, 1993.

[15] S. W. White, S. Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," D. J. Shippy, T. W. Griffith, "POWER2 Fixed–point, Data Cache, and Storage Control Units," T. N. Hicks, et al, "POWER2 Floating–point Unit: Architecture and Implementation," J. I. Barreh, et al, "POWER2 Instruction Cache Unit," *IBM Journal of Research and Development*, Vol. 38, No. 5, September, 1994, pp. 493-544.

[16] C. B. Stunkel, et al, "The SP1 High–Performance Switch," *Proceedings of the Scalable High Performance Computing Conference*, May, 1994, pp. 150-157.

[17] C. B. Stunkel, et al, "The SP2 Communication Subsystem," <URL:http://ibm.tc.cornell.edu/ibm/pps/doc/>.

[18] C. B. Stunkel, et al, "Architecture and Implementation of Vulcan," *Proceedings of the 8th International Parallel Processing Symposium*, April, 1994, pp. 268-274.

[19] F. H. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Technical Report UCRL-5375, Lawrence Livermore National Laboratory, December 1986.

[20] R. Hockney, "Performance Parameters and Benchmarking of Supercomputers," *Parallel Computing*, Vol. 17, No. 10 & 11, December, 1991, pp. 1111-1130.

[21] E. H. Welbon, et al, "The POWER2 Performance Monitor," IBM Journal of Research and Development, Vol. 38, No. 5, September 1994, pp. 545-554.